On freezeing and reactivating learnt clauses

Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, Lakhdar Saïs

On Freezing and Reactivating Learnt Clauses

Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs *

Université Lille-Nord de France
CRIL - CNRS UMR 8188
Artois, F-62307 Lens
{audemard,lagniez,mazure,sais}@cril.fr

Abstract. In this paper, we propose a new dynamic management policy of the learnt clause database in modern SAT solvers. It is based on a dynamic freezing and activation principle of the learnt clauses. At a given search state, using a relevant selection function, it activates the most promising learnt clauses while freezing irrelevant ones. In this way, clauses learned at previous steps can be frozen at the current step and might be activated again in future steps of the search process. Our strategy tries to exploit pieces of information gathered from the past to deduce the relevance of a given clause for the remaining search steps. This policy contrasts with all the well-known deletion strategies, where a given learned clause is definitely eliminated. Experiments on SAT instances taken from the last competitions demonstrate the efficiency of our proposed technique.

1 Introduction

The SAT problem, i.e. the problem of checking whether a set of Boolean clauses is satisfiable or not, is central to many domains of computer science and artificial intelligence (theorem proving, planning, non-monotonic reasoning, VLSI correctness checking or knowledge-base verification and validation). During the last two decades, SAT has gained considerable audience with the advent of a new generation of SAT solvers that are able to solve large instances encoding real-world applications. These solvers, called CDCL (Conflict Driven, Clause Learning) [11, 5], are based on a nice combination of (i) clause learning [9, 10, 15], (ii) VSIDS heuristics [11] and (iii) restart policies [6, 7], enhanced with efficient data structures (eg. Watched literals). On the theoretical side, K. Pipatsrisawat and A. Darwiche [13] proved that modern SAT solvers formalized as a proof system are equivalent in strength to general resolution, if the search is restarted at each conflict. This result shows that resolution-based clause learning is an important component of modern SAT solvers, since it pushes forward DPLL-like procedures from tree-like to general resolution, a more powerful proof system. On the practical side, as the set of clauses that can be derived from conflicts is of exponential size in the worst case, several strategies have been designed to cope with this combinatorial explosion problem. To maintain a learnt clause database of polynomial size - and consequently perform unit propagation with reasonable cost - all these strategies dynamically reduce the learnt database by deleting clauses considered to be irrelevant to the next search

* This work is (partially) supported by ANR UNLOC project: ANR 08-BLAN-0289-01.

steps. The most popular strategy considers a learnt clause as irrelevant if its activity or its involvement in recent conflict analysis is marginal. In [2], a static measure called literal block distance (LBD, corresponding to the number of different levels involved in a given learnt clause) is used to quantify the quality of learnt clauses. Clauses with smaller LBD are considered as more relevant. Theoretically, the first unique implication point (UIP) scheme is shown to be optimal among schemes that learn an asserting clause in terms of LBD measure [1]. The main drawback of these cleaning strategies is that they cannot avoid the elimination of relevant learnt clauses. Their irreversible elimination makes it possible that the same clause will be derived reapeatedly.

The problem of determining what is a useful learnt clause in advance remains very challenging and computationally hard. In this paper, we propose a new dynamic management policy of the learnt clause database in modern SAT solvers. It is based on a dynamic freezing and activation principle of the learnt clauses. At a given search state, it activates the most promising learnt clauses while freezing irrelevant ones. In this way, previously learned clauses can be discarded for the current step, but may be activated again in future steps of the search process. Our policy tries to exploit pieces of information gathered from the past to deduce the relevance of a given clause for the remaining search steps. This policy contrasts with all well-known deletion strategies, where a given learned clause is definitely eliminated. In this way, a clause can be useless at a given step and relevant at another step of the search process. The ideal is to freeze a learnt clause when it is not used and just to reactivate it at the time when it could play a role in the proof.

The next part of the paper is organized as follows: section 2 introduces necessary background. In section 3, we introduce a new relevance measure of learnt clauses, whereas in section 4, we present our dynamic freezing and activation strategy of learnt clauses. Before concluding, we present in section 5, an experimental comparison of our new dynamic learnt clauses management policy with the well known state-of-the-art reduction policies as well as state of the art solvers.

2 Definitions, notations and technical background

In this section, after some preliminary definitions and notations, we introduce the most salient computational features of modern SAT solvers.

A CNF *formula* Σ is a conjunction (interpreted as a set) of *clauses*, where a clause is a disjunction (interpreted as a set) of *literals*. A literal is a positive (x) or negative ($\neg x$) Boolean variable. The two literals x and $\neg x$ are called *complementary*. A *unit* clause is a clause with only one literal (called *unit literal*). An *empty clause*, is interpreted as false, while an *empty* CNF *formula*, is interpreted as true. A set of literals is *complete* if it contains one literal for each variable occurring in Σ and *fundamental* if it does not contain complementary literals. An *interpretation* $\mathcal{I}$ of a boolean formula Σ associates a value $\mathcal{I}(x)$ to some of the variables x appearing in Σ. An interpretation can be represented by a fundamental set of literals, in the obvious way. A *model* of a formula Σ is an interpretation $\mathcal{I}$ that satisfies the formula, *i.e.* that satisfies all clauses of the formula. Finally, SAT is the problem of deciding whether a given CNF formula Σ admits a model

Algorithm 1: CDCL solver

```
   Input: a CNF formula Σ
   Output: SAT or UNSAT
1  Δ = ∅;                                        /* learnt clause database */
2  while (true) do
3      if (!propagate()) then
4          if ((c = analyzeConflict()) == ∅) then return UNSAT;
5          Δ = Δ ∪ {c};
6          if (timeToRestart()) then backtrack to level 0;
7          else
8              backtrack to the assertion level of c;
9      else
10         if ((l = decide()) == null) then return SAT;
11         assert l in a new decision level;
12         if (timeToReduce()) then clean(Δ);
```

or not.

Let us now briefly describe the basic components of CDCL based SAT solvers [11, 5]. To be exhaustive, these solvers incorporate unit propagation (enhanced by efficient and lazy data structures), variable activity based heuristic, literal polarity phase, clause learning, restarts and a learnt clause database reduction policy.

These main components are depicted by the general scheme given in Algorithm 1. At each step of the main loop, the algorithm performs unit propagation (line 3). In case of conflict (lines 4-8), a new asserting clause is derived by conflict analysis (line 4). If such a clause is empty, then the formula is answered unsatisfiable, otherwise it is added to the learnt clause database (line 5). If it is not time to restart, the algorithm backjumps to the assertion level of the learnt clause, *i.e.* the level where the learnt clause becomes unit (line 8), otherwise it backjumps to the root of the search. When the formula is closed under unit propagation without generating the empty clause, a new decision literal - if it exists - is selected and asserted in a new decision level (line 11), otherwise a model is found and the formula is answered to be satisfiable (line 10).

Finally, when it is time to reduce, the learnt clause database is cleaned (line 12). This component, usually omitted in the description of CDCL solvers, is clearly crucial to the solvers' performance. Indeed, keeping too many learnt clauses will slow down the unit propagation process, while deleting too many of them will break the overall learning benefit. Consequently, identifying good learnt clauses - relevant to the proof derivation - is clearly an important challenge. The first proposed quality measure follows the success of the activity based VSIDS heuristic. More precisely, a learnt clause is considered relevant to the proof, if it is involved more often in recent conflicts, *i.e.* usually used to derive asserting clauses. Clearly, this deletion strategy supposes that a useful clause in the past could be useful in the future. More recently, a more accurate measure called LBD is used to estimate the quality of a learnt clause leading to a better

cleaning strategy than the previous one [2]. This new measure is based on the number of different decision levels appearing in a learnt clause and is computed when the clause is learnt. Extensive experiments demonstrates that clauses with small LBD values are used more often than those with higher LBD ones.

Another feature of CDCL solvers recently proposed in [12] concerns the literal polarity to be chosen when the next decision variable is selected thanks to the VSIDS heuristic. Usually, a default polarity (e.g. false) is defined and used each time a decision literal is assigned. Based on the observation that restarts and backjumping might lead to repetitive solving of same subformulas, Pipatsrisawat and Darwiche [12] proposed to dynamically save for each variable the last used polarity. This literal polarity based heuristic, called progress saving, prevents the solver from solving the same satisfiable subformulas several times. These memorized polarities can be represented as a complete interpretation $\mathcal{P}$. Each time a decision variable is chosen, its assignment polarity is selected from $\mathcal{P}$. Consequently, $\mathcal{P}$ gives us at least the polarities of the decision literals. And each time a literal is assigned by the solver, its associated polarity is set in $\mathcal{P}$.

In the next section, we exploit $\mathcal{P}$ (progress saving) to approximate the usefulness that one can expect in the near future from a learnt clause, in other words to measure the likelihood for a given clause to be part of the implication graph.

3 A new measure for identifying relevant learnt clauses

As mentioned above a CDCL-based SAT solver can be formulated as a resolution proof system [13, 3]. In practice, the main problem behind resolution-based techniques arises from their exponential space complexity. Consequently, the practical incarnation of modern SAT solvers can be seen as a resolution-based procedure with a deletion strategy. As a consequence, the completeness of modern SAT solvers is heavily connected to both the chosen deletion and restart policies. For example, if we use a restart with a static cutoff value and an aggressive deletion strategy, we cannot guarantee the completeness of the solver. For this reason one needs to be careful when designing a deletion strategy. Consequently, defining what is a relevant clause before completing the proof itself is of a great importance for the efficiency of the solver. However answering such a question is computationally hard and it is related to finding a proof of minimal size.

In this section we define a simple measure to identify the relevance of a given learnt clause and we experimentally show its effectiveness. Our measure is based on the progress saving polarity [12] introduced in the previous section. This *progress saving based quality measure*, in short psm, is defined as follows: given a clause c and a complete set of literals $\mathcal{P}$ representing the current set of saved literals polarities, we define $psm_{\mathcal{P}}(c) = |\mathcal{P} \cap c|$. This measure can be related to another one proposed in [11]. In this paper, a learnt clause was tagged useless, in the goal to delete it, if its number of unassigned literals has reached a predefined threshold.

First let us note that the *psm* measure is highly dynamic. Since the set $\mathcal{P}$ of saved literals polarities will evolve during search, the *psm* of a given clause will also evolve consequently. For example, when a clause is learnt, its *psm* value is equal to zero and becomes one after backjumping to the assertion level. It is also important to note that

when a given learnt clause is at the origin of unit propagation, its *psm* value is also equal to one. These preliminary remarks suggest that clauses with small *psm* value are the most relevant to the near future of the search. Let $\mathcal{I}$ be the current partial interpretation and $\mathcal{P}$ be the current complete interpretation representing the current saved literals polarities, and c a learnt clause. As $\mathcal{I} \subset \mathcal{P}$, $psm_{\mathcal{P}}(c)$ represents the number of literals that are assigned to true by $\mathcal{I}$ or that would be assigned to true by $\mathcal{P} \backslash \mathcal{I}$. Consequently, a clause with a small *psm* value has a lot of chance to be unit propagated or to be falsified. On the contrary, a clause with a big *psm* value has a lot of chance to be satisfied by more than one literal and then to be irrelevant for the subsequent part of the search.

To analyze and to validate this assumption, experiments are conducted on some SAT instances. Figure 1 shows, for a sample of instances, the average number of times clauses with a given *psm* value are used during the unit propagation process. In this experiment, we consider a time sequence t_k with $k > 0$ (the search starts at t_0) corresponding to the successive steps of the search where the learnt database is classically reduced. Let $\mathcal{P}_{t_k}$ and $\mathcal{P}_{t_{k+1}}$ be the progress saving literal polarities at the steps t_k and t_{k+1} respectively. Let us consider the time window between t_k and t_{k+1}, when a given clause c from the learnt database is used for unit propagation, we compute $psm = psm_{\mathcal{P}_{t_k}}(c)$ and then $\alpha(psm)$ the number of times a clause with such psm value is used for propagation is increased by one. The average number of times a clause with a given psm value (x-$axis$) is used in unit propagation (y-$axis$), corresponds to $\alpha(psm)$ divided by the total number of times a learnt database is reduced.

As we can observe from Figure 1, learnt clauses with small *psm* value are used more often in the unit propagation process than those with higher *psm* value. If we look closer, we can see that the most used clauses are those with psm value around 10. Based on extensive experiments, we observed that on the majority of instances the distribution of psm values looks like those represented in the two upper curves of Figure 1.

This first experiment illustrates the relevance of clauses with small psm value. To compare it with previous learnt clauses quality measure, we integrate our psm measure to the learnt clauses reduction policy (clean(Δ) - line 12) of MINISAT 2.2 which is the latest version of the well known solver MINISAT [5]. Similarly to previous approaches, each time a reduction is performed, the set of clauses is sorted according to the increasing order of psm value. When two clauses admit the same psm value, the one with the greatest activity (VSIDS) is preferred. Then the learnt database is reduced by half. Like other strategies, we keep the binary clauses in the learnt database.

In the sequel, all our experiments are conducted on a Quad-core Intel XEON X5550 with 32Gb of memory, using the 292 application instances of the SAT 2009 competition. The CPU time limit is set to 900 seconds.

For each solver, we indicate the number of solved instances (#Solved) with the number of satisfiable (#SAT) and unsatisfiable instances (#UNSAT) in brackets. We also give the average time in seconds (avg time) necessary to solve these instances.

Table 1 summarises the results obtained by MINISAT^d [5], MINISAT^d + LBD [2] and MINISAT^d + psm using the default time sequence (noted MINISAT^d) of MINISAT. As we can see, MINISAT^d + psm obtains the best overall results and is the best on satisfiable instances. This first experiment shows the efficiency of our new measure *psm* using the default time reduction sequence of MINISAT.

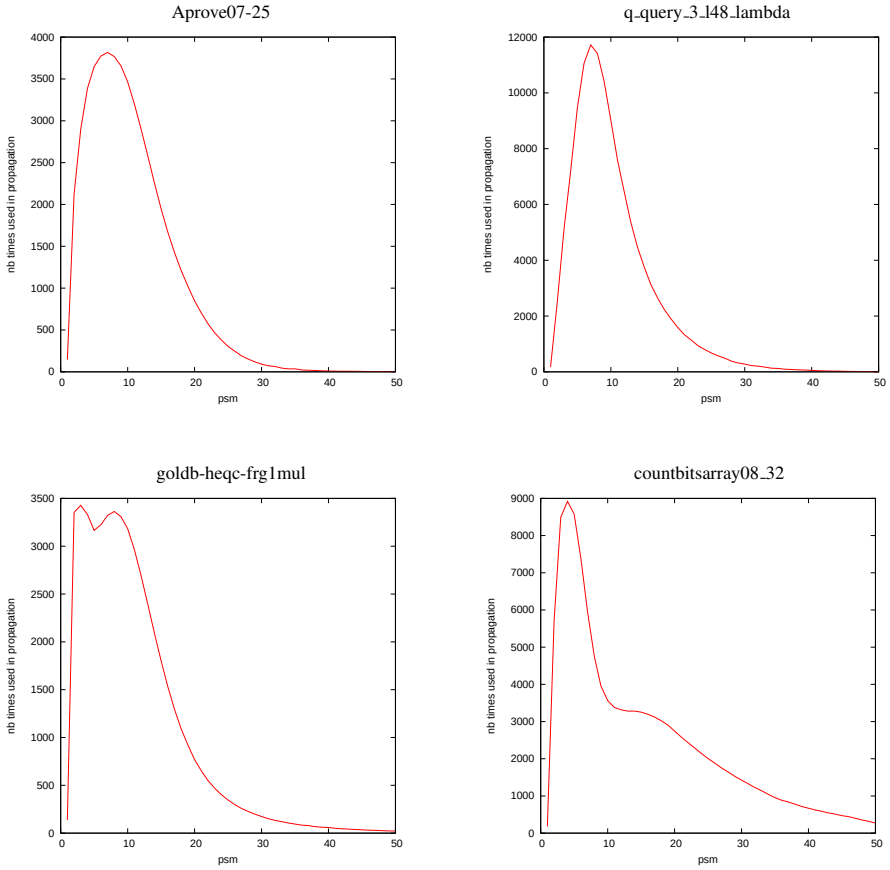

Fig. 1. Progress saving measure / relevance with respect to UP

To make a fair comparison between these three approaches, we also present in Table 2 the results obtained using an aggressive cleaning policy as presented in [2] (noted MINISAT^a). In this experiment the learnt database Δ is reduced using the following time sequence, $t_0 = 4000$ conflicts and $t_k = t_{k-1}+300$ conflicts for $k > 0$. Using aggressive (more frequent) cleaning time sequence, the result obtained by the LBD measure are better than those obtained by VSIDS like criterion and the *psm* measure.

As a summary, considering the classical reduction and deletion strategies, these first experiments clearly show that our measure is competitive with the two other well-known measures using both aggressive and less aggressive cleaning policy. This measure will be used in next section in order to design a dynamic managment policy of learnt clauses.

Solver	#Solved (#SAT- #UNSAT)	avg time
MINISAT^d	174 (68 - **106**)	142
MINISAT^d + *psm*	**177** (**73** - 104)	130
MINISAT^d + LBD	173 (71 - 102)	132

Table 1. Results with the MINISAT default time cleaning sequence.

Solver	#Solved (#SAT- #UNSAT)	avg time
MINISAT^a	162 (68 - 94)	136
MINISAT^a + *psm*	163 (70 - 93)	140
MINISAT^a + LBD	**168** (**72** - **96**)	128

Table 2. Results with an aggressive time cleaning sequence.

4 Freeze and reactivate: a dynamic management policy

In section 3, we defined a new measure based on progress saving [12] for identifying relevant learnt clauses. In this section, we describe our dynamic management policy of the learnt clause database. Our proposed framework is based on two important key points. First, the progress saving based measure is highly dynamic and evolves during search. Consequently, a clause might be considered irrelevant (high *psm* value) at a given step of the search and could become relevant (small *psm* value) in the future steps of the search. Secondly, determining if a given learnt clause will be involved again in the resolution proof is a computationally hard task. All the well-known management policies are not safe from regularly eliminating relevant learnt clauses. For both reasons, our proposed approach introduces an additional and new concept of frozen learnt clauses. A learnt clause considered as irrelevant at a given step can be frozen and reactivated when it is considered as useful again. More precisely, freezing (respectively activating) a clause means that the clause is disconnected (respectively attached) to the learnt database, and then it is not used during the search (respectively used).

This kind of management strategy cannot be defined using the other known measures such as activity and LBD-based ones. Indeed, the LBD value of a given clause is definitely set at the time of its generation and does not change during search, while the activity (VSIDS-based) measure is dynamic but can only be used to update the activity of learnt clauses currently in the database.

Let us now formally describe our new learnt clause management policy. First, as the *psm* value of a given clause is highly dynamic, we introduce a notion of deviation between two successive sets of progress saving polarities. Let V_{t_k} be the set of variables assigned by the solver between two consecutive time sequences (as defined in previous section) t_{k-1} and t_k. The deviation d_{t_k} is defined as follows: $d_{t_k} = \frac{h(\mathcal{P}_{t_k}, \mathcal{P}_{t_{k-1}})}{|V_{t_k}|}$, where h is the usual hamming distance.

This deviation defined as a normalized hamming distance, gives us an outline of the evolution of progress saving polarities between two successive cleanings of the learnt database. A deviation tending to zero indicates that the solver explores around the same part of the search space whereas a value close to one indicates that the solver explores different part of the search space.

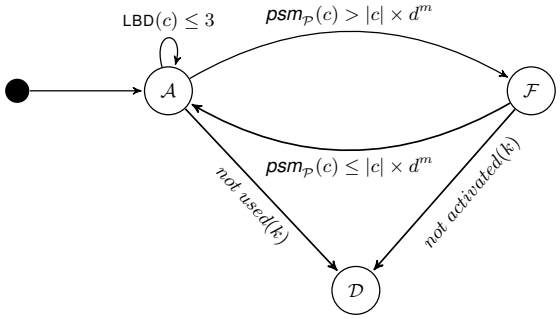

Fig. 2. State diagram of a learnt clause.

To obtain a more precise view of the search behavior, we introduce another notion of minimal deviation $d^m_{t_k} = min\{d_{t_i} | 0 \leq i \leq k\}$ at time step t_k.

Using this minimal deviation, we can now refine our psm measure. Indeed, let c be a clause to be evaluated at time step t_k, if $psm_{\mathcal{P}_{t_k}}(c) > d^m_{t_k} \times |c|$ then the clause c is likely to be satisfied in a near future, otherwise it is likely to be involved in the propagation process.

Our approach depicted in Figure 2 is represented as a state diagram. At each cleaning t_k, learnt clauses can move from a state to another one following some conditions.

First, a learnt clause c can be in one of the three following states:

1. *Active state* $\mathcal{A}$: c is active and watched.
2. *Frozen state* $\mathcal{F}$: c is frozen i.e. c is not watched
3. *Dead state* $\mathcal{D}$: c is deleted.

Let us describe these different transitions:

- Each time a clause is learnt it enters the state $\mathcal{A}$.
- A clause $c \in \mathcal{A}$ with a short LBD ($lbd(c) \leq 3$ in the figure) remains in the state $\mathcal{A}$ until the end of the search process.
- A clause $c \in \mathcal{A}$ such that $\frac{psm_{\mathcal{P}_{t_k}}(c)}{|c|} > d^m_{t_k}$ enters the frozen state $\mathcal{F}$.
- A clause $c \in \mathcal{F}$ such that $\frac{psm_{S_{n_i}}(c)}{|c|} \leq d^m_{n_i}$ enters the active state $\mathcal{A}$.
- A clause $c \in \mathcal{F}$ not activated after k time steps is deleted. Similarly, a clause $c \in \mathcal{A}$ remaining active more than k steps without participating to the search is also deleted. In both cases, it enters the state $\mathcal{D}$ after $k = 7$ time steps in our experiments.

One of the main advantages of our approach comes from the fact that we can perform frequent cleaning of the learnt clause database without taking care of removing relevant clauses. So we choose a very aggressive policy. We set $t_0 = 500$ conflicts, and $t_k = t_{k-1} + 100$ conflicts.

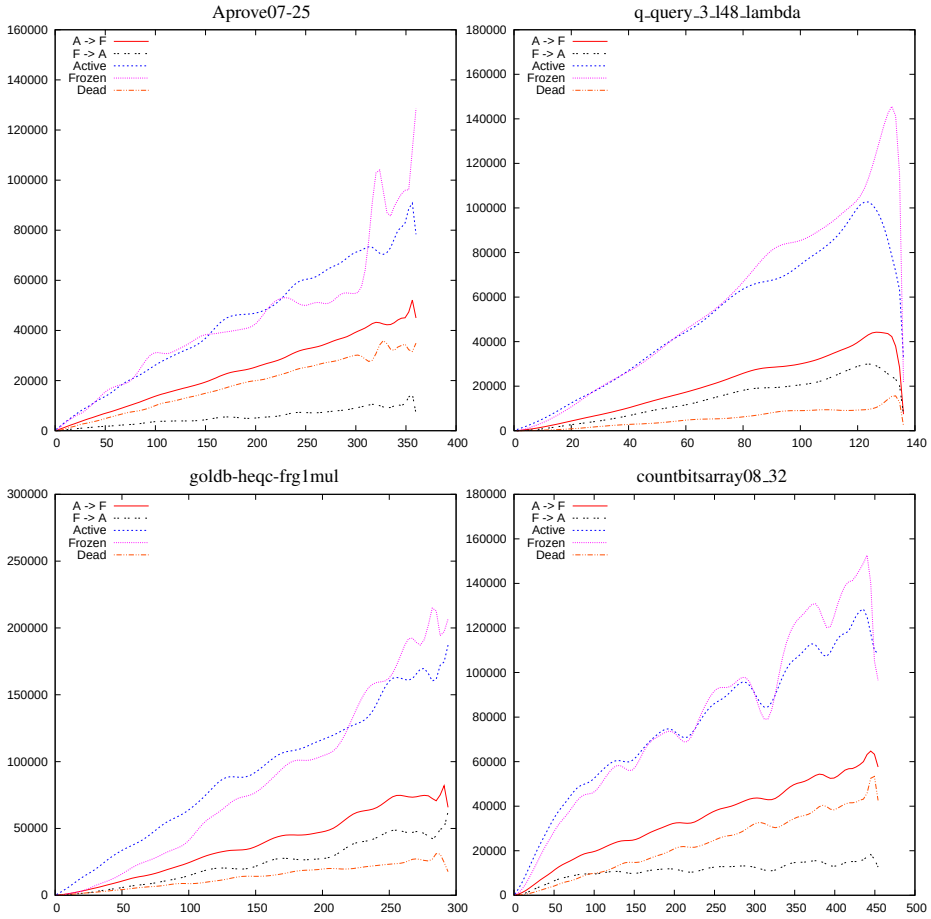

Fig. 3. Evolution of the number of clauses in different states and number of state transfers

We conducted some experiments to analyse the transfer of the clauses from the state $\mathcal{A}$ to the state $\mathcal{F}$ and *vice versa*. Figure 3 shows, for the same sample of instances as in the Figure 1, the number of deleted clauses, the number of transitions to the frozen state, the number of transitions to the active state, the number of active (or watched) learnt clauses and finally the number of frozen clauses. These data are represented by the y-axis, whereas the x-axis represents the cleaning operated at the time step t_k. For clarity reasons, all curves have been smoothed. For all instances, the number of frozen clauses ($Frozen$) and the number of active clauses ($Active$) are relatively similar. The curve representing the number of clauses becoming active ($\mathcal{F} \rightarrow \mathcal{A}$) is dominated by those representing the number of clauses becoming frozen ($\mathcal{A} \rightarrow \mathcal{F}$). However, the two curves evolve similarly and they are closer on some instances (e.g. $q\_query\_3\_l48\_lambda$) than on others (e.g. $Aprove07 - 25$) . Finally, we can also observe that, at each cleaning time step, some clauses are definitively deleted ($Dead$).

5 Empirical evaluation

This section is divided in two parts. In the first, we compare our dynamic management policy (psm_{dym}) against the classical reduction approach with different quality measures (LBD, VSIDS like, *psm*). In the second experiment, we compare it with three state-of-the-art solvers: GLUCOSE which embeds LBD measure, a dynamic restart policy and some other features [2], LINGELING which also embeds more powerful reasoning like blocked clause elimination [8], and finally, CRYPTOMINISAT which adds many other features (e.g. vivification, reasoning on xor clauses...). Descriptions of these solvers are available on the SATRACE 2010 website `http://baldur.iti.uka.de/sat-race-2010`. Except for LINGELING and CRYPTOMINISAT which embed preprocessing inside, the other solvers use `SatElite` for preprocessing [4].

In the first experiment, we use the same solver and the only difference is in the learnt clause management policy. In the second experiment, our aim is to compare our learnt clause management approach integrated in MINISAT 2.2 (MINISAT-psm_{dyn}) with the state-of-the-art SAT solvers. Source code and extensive experiments can be found at http://www.cril.fr/~lagniez/ressource.html.

5.1 Comparison with different quality measures

We compare our dynamic policy, called MINISAT-psm_{dyn} with the classic MINISAT, and MINISAT with learnt database reduction based on *psm* (MINISAT-*psm*) and on LBD (MINISAT-LBD) (like in section 3). Figure 4 summarizes the results. It contains three scatter plots corresponding to the comparison of MINISAT-psm_{dyn} with the 3 others solvers. In such a plot, each dot corresponds to a given instance, the x-axis corresponds to the cpu time needed by the MINISAT, LBD or *psm* to solve the instance, whereas the y axis corresponds to the cpu time needed by psm_{dyn} to solve it. So, dots below the diagonal correspond to instances solved faster by MINISAT-psm_{dyn} (SAT and UNSAT instances are differentiated). Figure 4 also contains a cactus plot related to the comparison of the 4 solvers.

It is quite clear that our freezing strategy outperforms the other strategies. It solves 189 instances (76 SAT and 113 UNSAT), which is significantly better than the other solvers (see Table 1). Furthermore, as we can see on the scatter plots, MINISAT-psm_{dyn} solves instances faster than the others solvers.

5.2 Comparison with state of the art solvers

Figure 5 summarizes the comparison with state of the art solvers. It is structured as figure 4. Let us detail the number of solved instances by each solver: LINGELING solves 187 instances (77 SAT, 110 UNSAT), GLUCOSE 189 (70 SAT and 119 UNSAT) and CRYPTOMINISAT 194 (74 SAT, 120 UNSAT). These results and the plots of Figure 5 show that our dynamic management policy is really competitive with state-of-the-art solvers (remember, it solves 189 instances (76 SAT and 113 UNSAT)). It does not even embed sophisticated components such as dynamic restart, etc.

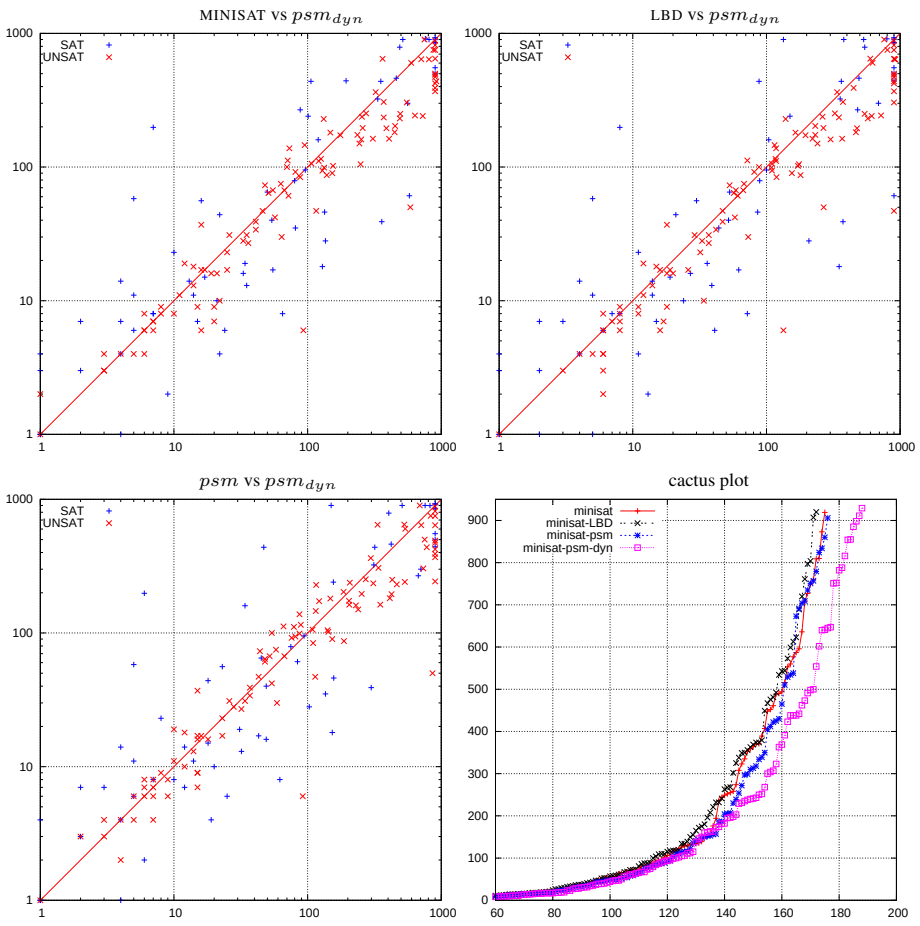

Fig. 4. Comparison with different learnt clauses quality measures.

6 Conclusion

In this paper, we introduced a new measure for identifying relevant learnt clauses. The main advantage of this measure is that it is dynamic (unlike the LBD measure) and it can be computed even if clauses do not participate in the search process (unlike the VSIDS like measure). Thanks to this property, a new learnt clause database management framework has been proposed. It exploits a novel dynamic policy that activates the most promising learnt clauses while freezing irrelevant ones. This is in contrast with all the well-known deletion strategies, where a given learned clause is definitely eliminated. Experiments on SAT instances taken from the last competitions demonstrate the effectiveness of our approach.

As future work, we plan to exploit the evolution of the set of progress saving literal polarities in order to decide if cleaning has to be performed. Considering the connection

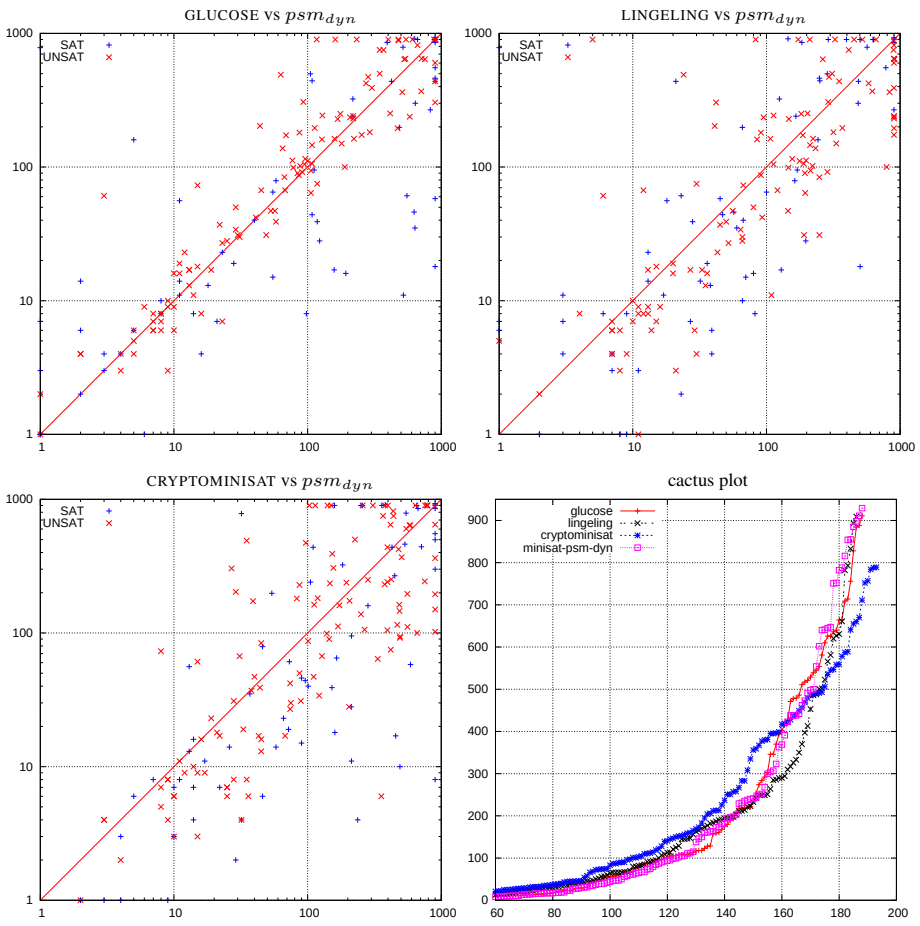

Fig. 5. Comparison with state of the art solvers: GLUCOSE, LINGELING and CRYPTOMINISAT.

between restarts and clause learning [14], we plan to exploit this connection to improve our proposed leant database management approach.

Aknowledgements We would like to thank the anonymous reviewers for insightful comments

References

1. Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Saïd Jabbour, and Lakhdar Saïs. A Generalized Framework for Conflitcs Analysis. Technical Report MSR-TR-2008-34, Microsoft Research, 2008.
2. Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *proceedings of IJAI*, pages 399–404, 2009.

3. Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.
4. N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *proceedings of SAT*, pages 61–75, 2005.
5. Niklas Een and Niklas Sörensson. An extensible SAT-solver. In *proceedings of SAT*, pages 502–518, 2003.
6. Carla Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *proceedings of AAAI*, pages 431–437, 1998.
7. Jimbo Huang. The effect of restarts on the efficiency of clause learning. In *proceedings of IJCAI*, pages 2318–2323, 2007.
8. Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In *proceedings of TACAS*, pages 129–144, 2010.
9. Roberto J. Bayardo Jr. and Robert Schrag. Using csp look-back techniques to solve real-world sat instances. In *proceedings of AAAI*, pages 203–208, 1997.
10. J. Marques-Silva and K. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *proceedings of ICCAD*, pages 220–227, 1996.
11. Matthew Moskewicz, Connor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *proceedings of DAC*, pages 530–535, 2001.
12. Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *proceedings of SAT*, pages 294–299, 2007.
13. Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning sat solvers with restarts. In *proceedings of CP*, pages 654–668, 2009.
14. Knot Pipatsrisawat and Adnan Darwiche. Width-based restart policies for clause-learning satisfiability solvers. In *proceedings of SAT*, pages 341–355, 2009.
15. Lintao Zhang, Connor Madigan, Matthew Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *proceedings of ICCAD*, pages 279–285, 2001.